Volume 2 No. 3, March 2012 ISSN 2223-4985

International Journal of Information and Communication Technology Research

http://www.esjournals.org

Development of a Scalable Architecture for Dynamic Web-Based Applications

Oluwagbemi Oluwatolani[1] , Afolabi Babajide[2], Achimugu Philip[3]
[1,3]Department.of Computer and Information Science, Lead City University, Ibadan.
[2]Department.of Computer Science and Engineering, Obafemi Awolowo University, Ile-Ife.

ABSTRACT

Web-based applications play a major role in the development and deployment of online services. Due to their unique medium of operation, they also have unique requirements when it comes to data storage. Most web applications use databases as their primary data storage mechanism, which does not only provide well-developed routines for data management, but also can be accessed via common interfaces such as Microsoft's Object Linking and Embedding Database (OLEDB) and Sun's JAVA Database Connectivity (JDBC). Scalability poses a significant challenge for today's web applications, mainly due to the large population of potential users. As a result, we attempt to present a scalable architectural model that could be adopted by web application developers in order to implement scalable systems with good quality of service. This will ensure that required information is available during critical times irrespective of sudden load increase.

Keywords: *Online services, Scalability, Database, Web Application, architecture*

1. INTRODUCTION

The world-wide web has taken an important place in everyday's life. Many businesses rely on it to provide their customers with immediate access to information. However, to retain a large number of customers, it is important to guarantee a reasonable access performance regardless of the request load that is addressed to the system. Web application hosting systems therefore need the ability to scale their capacity according to business needs.

The Web is increasingly becoming more "dynamic", the content is produced by programs that execute at the time a request is made and is often customized based on several factors like a user's preferences and the previous content the user has viewed. Dynamic content allows creation of rich interactive applications like social networks, bulletin boards, civic emergency management, and e-commerce applications, which represent the future landscape of Web applications.

Web applications are typically deployed on a three-tiered server-side architecture consisting of one or more instances of: *a web server*, *an application server*, and *a database server* [1]. The web server manages the HTTP (Hyper-Text Transfer Protocols) interactions, the application server runs the application code, and the database server houses the application's database. All these servers are referred to as home *server(s)*. (Figure 1 shows the resulting architecture.)

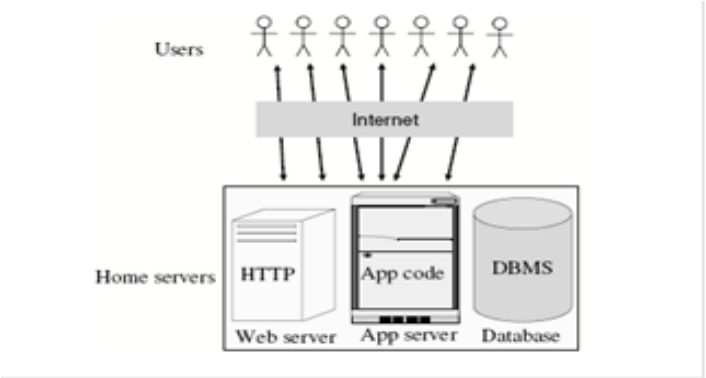

Figure 1: Traditional Web-Based Architecture
Source: [1]

The key to scalability is to ensure that the home server(s) remains lightly loaded even at high request rates. Because an application's web servers and application servers do not carry any persistent state, they can be replicated so that each replica remains lightly loaded even at high request rates. Alternatively, an application can use a Content Distribution Network that executes application code to scale its web and application server.

The main challenge is to design a *Database Scalability Service (DBSS)*, which can effectively offload at least some of the database work from the home infrastructure's database server(s) into an interoperable rescue database system.

More so, many research efforts have been made to provide scalable infrastructures for static content. However, scaling web applications that dynamically generate content remains a challenge. This research therefore, seeks to propose a

framework that will enhance good quality of service from web applications regardless of increase in users.

2. SCALABILITY CHALLENGES IN WEB-BASED APPLICATIONS

Dynamic Web applications are characterized by capabilities for personalization and distributed updating of data. These features, coupled with the often sensitive nature of the associated data, create new systems and security challenges in building an effective scalability service for dynamically-generated Web content. Web applications are typically deployed on a three-tiered server-side architecture consisting of one or more instances each of: a web server, an application server, and a database server. Most advanced Web applications rely on the database server(s) for the bulk of the data management tasks, and indeed the database servers often become the bottleneck in terms of maximum supportable load. Consequently, an effective scalability service would have to offload at least some of the database work from the home organization. However, that task is encumbered by two major difficulties inherent in dynamic Web applications:

a. Most advanced Web applications require strong consistency for their most important data. It is well-known that maintaining strong consistency among replicas in a distributed setting presents significant scalability challenges.
b. Administrators are typically reluctant to cede ownership of data and permit data updating to take place outside the home organization. This reluctance arises with good reason, due to the security concerns, data corruption risks, and cross organizational management difficulties entailed. Difficulty firstly precludes caching techniques based entirely on timed data expiration, i.e., *time-to-live* (TTL) protocols [2], which are the norm in current approaches to scaling the delivery of static Web content. The growing number of dynamic Web applications with sensitive and mission-critical data requires a high degree of data fidelity and rely on systems that adhere to the transactional model of consistency. Secondly, in many cases, capabilities for data updating simply must remain within the boundaries of the home organization, and decentralized data updating is simply unacceptable.

3. LITERATURE REVIEW

In the past years, many techniques have been proposed to improve the scalability of web applications. The simplest one is *edgeserver computing* where requests are distributed among several edge servers running the same code ([3]; [4]). Although this technique is very effective at scaling the computation part of the applications, the main challenge is to scale the access to the application data. Replication is a common technique to improve the throughput of a Relational Database Management Systems (RDBMS). Many RDBMS replication solutions aim at replicating data across multiple servers within a cluster ([5]; [6]; [7]; [8]). Database replication allows one to distribute read queries among the replicas. However, in these solutions, all Universal Description Interface (UDI) queries must first be executed at a master database, then propagated and re-executed at all other "slave" databases using 2-phase commit or snapshot isolation mechanisms. A few commercial database systems such as Oracle allow one to optimize the re-execution of UDI queries at the slaves by transferring a log of the execution at the master. However, these techniques do not improve the maximum throughput as they require a single master server to execute all UDI queries. The throughput of the master server then determines the total system's throughput. As extensively discussed in [9], a number of techniques have been developed specifically to scale Web applications. Most of these techniques exploit the fact that Web applications issue a fixed number of query templates to the database. Several systems propose to cache the result of database queries at the edge servers. Static knowledge of conflicts among different query templates allows one to invalidate caches efficiently when a database update occurs. However, such systems work best under workloads with high query locality, few UDI queries, and few dependencies between query templates. Furthermore, the efficiency of caches does not grow linearly with the quantities of resources assigned to it, so caches alone cannot provide arbitrary scalability. Another approach based on query templates relies on partial database replication, where not all tables get replicated to all database servers [10]. This allows one to reduce the number of UDI queries that must be issued to each database server. However, although this technique allows one to improve the system throughput, its efficiency is constrained by the templates that query multiple tables simultaneously (because of join queries or database transactions). In [11], the authors propose an edge computing infrastructure where the application programmers can choose the best suited data replication and distribution strategies for the different parts of application data. By carefully reducing the consistency requirements and selecting the replication strategies, this approach can yield considerable gains in performance and availability. However, it requires that the application programmers have significant expertise in domains such as fault-tolerance and weak data consistency. Several recent academic and industrial research efforts have focused on the design of specialized data structures for scalable and highly available services ([12], [4]; [13]). These systems usually provide a simple key-based put/get interface and focus on scalability and availability properties rather than rich functionality or transactional properties. These design choices mean that these systems implicitly or explicitly assume that an application has been decomposed into separate entities with very simple data access patterns. Data fragmentation techniques have been commonly used in the design of distributed relational database systems ([14]; [15]). In these works, tables are partitioned either vertically or horizontally into smaller fragments. Partitioning schemes are determined according to a workload analysis in order to optimize access time. However, these techniques do not fundamentally change the structure of the data, which limits their efficiency. Furthermore, changes in

the workload require to constantly re-evaluating the data fragmentation scheme [16].

In view of the aforementioned, our paper demonstrates how one can design a Web application along a service-oriented architecture with simplified workload.

4. PROPOSED SYSTEM DESIGN

Our approach exploits two properties shared by many Web applications: the underlying data workloads tend to (1) be dominated by reads, and (2) consist of a small, fixed set of query and update templates. The first property makes it feasible to handle all data updates at servers within each application's home organization. That way, no data updating is performed outside of the home organization, and tight control over authentication of updates and overall data integrity is retained. By exploiting the second property, that is, predefined query and update templates, it is believed a fully distributed mechanism for enforcing strong cache consistency that does not burden home organizations with this responsibility, can be created.

The architecture we envision for our scalability service is illustrated in Figure 2. At the bottom we see *home servers*, which host code and data for individual applications within the boundaries of their home organizations. Scalability is provided by a collection of interoperating *proxy servers* that cache data on behalf of home servers. Users access applications indirectly, by connecting to proxies within the scalability service network. At the proxy servers, application data is cached on a strictly read-only basis, ensuring that server failures do not affect data integrity or require expensive quorum protocols. All data update requests are forwarded directly to application home servers for local processing. Cache consistency is managed entirely by the proxy servers themselves, which notify each other of updates via a fully distributed multicast network.

4.1 Implementation Overview

The salient components of our prototype system are shown in Figure 2. We implemented our system in *php programming language* and the overall system was installed on a WAMP server. We use MySQL at the back end. Each proxy server runs a *logger service*, which is responsible for collecting information regarding which cluster was accessed by the Web clients. The logger is implemented as a stand-alone multithreaded server that collects information from the driver and periodically updates the access database. Also, as seen in the figure 2; the home server runs two special services, *clustering* and *replica placement* services. The clustering service performs the clustering of data units during the initial stages of system deployment. It periodically collects access patterns from the proxy servers and stores it in its database. Subsequently, it performs clustering with an embedded command in the algorithm. The home server only acts as backend database, so it does not need to have a Web server or logger service. The user profile component is responsible for allotting intending users according to their subject of need. While, the *replica placement* service in the home server finds the "best" locations for hosting a replica and master for each cluster. It does so by evaluating the cost obtained by different replica placement strategies and master selection for the cluster's access pattern in the previous period. Note that once data units are clustered, the logger service in the edge servers starts collecting access patterns at the cluster level. This information is then used by the replica placement service to place the replicas. Upon deciding which servers would host which clusters, the placement service builds the cluster property table for each cluster and copies it to all proxy servers.

To perform periodic adaptation, the replica placement service is invoked periodically. We note that the period of this adaptation must be set to a reasonable value to ensure good performance and stability. It is intended to study the need and support for continuous adaptation and effective mechanisms for them in the future. The clustering service is also run periodically.

4.2 Distributed Consistency Management

The distributed consistency mechanism implemented combines two technologies: query/update independence analysis and distributed multicast. Query/update independence analysis is concerned with deciding whether a given update affects the result of a particular query. Distributed multicast environments offer efficient distributed routing of messages to multiple recipients based on application-level concepts rather than network addresses. In our approach, application code is first analyzed statically to identify pairs of embedded query and update templates that are in potential conflict, meaning that an instantiation of the update template might affect the result of an instantiation of the query template. At runtime, when an update template is invoked at a proxy server, the precomputed set of potentially conflicting queries is narrowed based on the parameter bindings supplied with the update. At that point, it must be ensured that any cached results of conflicting queries are either updated or invalidated. Hence, all proxy servers caching conflicting data must be notified of the update. In order to limit dissemination of updates to just the servers that cache potentially conflicting data, our approach leverages distributed multicast technology. Proxy servers organize themselves into an overlay network, and transmission of updates is handled via a group multicast environment built into the overlay. Multicast groups are established in correspondence with query templates embedded in the application, and servers caching data for a particular query template subscribe to the corresponding multicast group(s). When an update occurs, notification of that update is routed by the multicast protocol to any and all servers caching data that may be affected.

4.3 Cache Invalidation

When a proxy server receives notification of an update that may conflict with locally-cached data, some action must be taken to ensure consistency with the master copy maintained by the home server(s). Since one of our design principles is to avoid distributed updating of data, we plan to rely on

invalidation, in which potentially inconsistent data is evicted from caches. In this design (Figure 2), each proxy server is equipped with a local *invalidator* module, which decides what data to evict in response to notification of an update. Cached data is never updated locally, so home organizations can more readily monitor and control the security and integrity of their data. Authentication of update requests need take place only at home servers. (While unauthenticated updates may induce spurious invalidation of cached data, the impact is only on performance, not correctness.)

There is an important tradeoff to be considered in the design of invalidator modules. Clearly, to achieve the best scalability, a minimal number of data invalidations should be performed by proxy servers. The most selective invalidation strategies require inspection of the content of cached data. However, the presence of data originating from databases managed by different Database Management Systems products severely complicates this process, and it is well known that increased implementation complexity tends to degrade reliability. In addition, allowing proxy servers to inspect the content of cached data precludes the use of certain cryptographic approaches for ensuring privacy and security. Efforts are being geared toward formally characterizing the tradeoff between scalability, on the one hand, and the combination of low implementation complexity and secrecy of data on the other hand.

A prototype scalability service for dynamic Web applications was built in order to enhance the performance of existing scalability model for web based applications. This prototype enables application providers to offload the dynamic generation of Web content to a distributed network infrastructure. In contrast with previous work on distributed generation of dynamic Web content, an approach that replicates all three tiers (web server, application server, back-end database) of the traditional centralized Web architecture was explored. The traditional architecture is illustrated in Figure 1 (here all three tiers are depicted as executing on a single home server; in general they may be spread across multiple physical servers). As in the centralized architecture, our distributed architecture includes a home server for each application provider, located within the provider's home organization. In the distributed architecture, the home server primarily houses code and data, while a set of proxy servers processes client requests and executes programs. A client request is sent to a proxy server which then generates the appropriate response by a running a program, accessing locally cached code and data when possible, and obtaining additional code and data from the home server as necessary. In the following subsections, the design choices made are described while building the prototype. The designs of the three distinct types of nodes in our system (i.e. home servers, proxy servers, and clients) are presented.

4.4 Home Servers

Each home server embodies the traditional three-tiered architecture, which enables it to generate Web content dynamically. While it is aimed at offloading the generation of dynamic content to the proxy servers, the prototype is compatible with this well-established home server architecture and hence also allows the application provider to serve directly as much dynamic content as it chooses. The top tier is a standard *web server*, which manages HTTP interactions with clients. The web server is augmented with a second tier, the *application server*, which can execute a program to generate a response to a client's request based on the client profile, the request header, and the information contained in the request body. Finally, the third tier consists of a *database server* that the application provider uses to manage all of its data.

4.5 Proxy Servers

The core of the prototype lies in the architecture and implementation of the proxy servers. A traditional proxy server is sufficient to serve static content on behalf of application providers. To generate dynamic content however, the system was designed in such a way that each proxy server has immediate and dynamic access to the contents in the main or home server. The proxy server also contains a simple database cache designed to reduce CPU utilization and bandwidth consumption of the home server, as well as the impact of database queries on the execution time of a request.

4.6 Clients

The clients are responsible for sending requests. The responses of these requests must be provided at good or expected time. Scaling application-specific computation is relatively easy as requests can be distributed across any number of independent application servers running identical code. Similarly, one can reduce the network bottleneck between the application and database servers. The main challenge, however, is to scale access to application data.

4.7 Caching Granularity

The proxy server caches the results of database queries (that is, materialized views) rather than the tables of the database itself, or arbitrary sub tables. The primary rationale for this design choice is to make the proxy independent of the back-end database implementable. This flexibility is required since different application providers may choose different back-end databases. This approach also has the advantage that complex queries need not be re-executed, and the proxy server does not have to implement full database functionality (for example, it does not need a query optimizer or query plan evaluator). In the prototype, database query results are cached at the *phpMyAdmin* level. All updates are forwarded directly to the back-end database at the home server.

4.8 Data Consistency

One important issue in any replicated system is consistency. Consistency management has two main aspects:

update propagation and concurrency control. In update propagation, the issue is to decide which strategy must be used to propagate updates to replicas. Many strategies have been proposed to address this issue. They can be widely classified into push-based and pull-based strategies. Pull-based strategies are mostly suitable for avoiding unnecessary data update transfers when no data access occurs between two subsequent updates. In this system, it was decided to use a *push* strategy where all updates to a data unit are pushed to the replicas immediately. Pushing data updates immediately ensures that replicas are kept consistent and that the servers hosting replicas can serve read requests immediately. Yet, propagating updates is not sufficient to maintain data consistency. The system must also handle concurrent updates to a data unit emerging from multiple servers. Traditional non-replicated DBMSs perform concurrency control using locks or multiversion models. For this, a database requires an explicit definition of *transaction*, which contains a sequence of read/write operations to a database. When querying a database, each transaction sees a snapshot of consistent data (a database version), regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data produced by concurrently running update transactions.

However, concurrency control at the database level can serialize updates only to a single database and does not handle concurrent updates to replicated data units at multiple edge servers. Traditional solutions such as two-phase commit are rather expensive as they require global locking of all databases, thereby reducing the performance gains of replication. To handle this scenario, the system must serialize concurrent updates from multiple locations to a single location. The system does guarantee transaction semantics and also enforces consistency for each query individually.

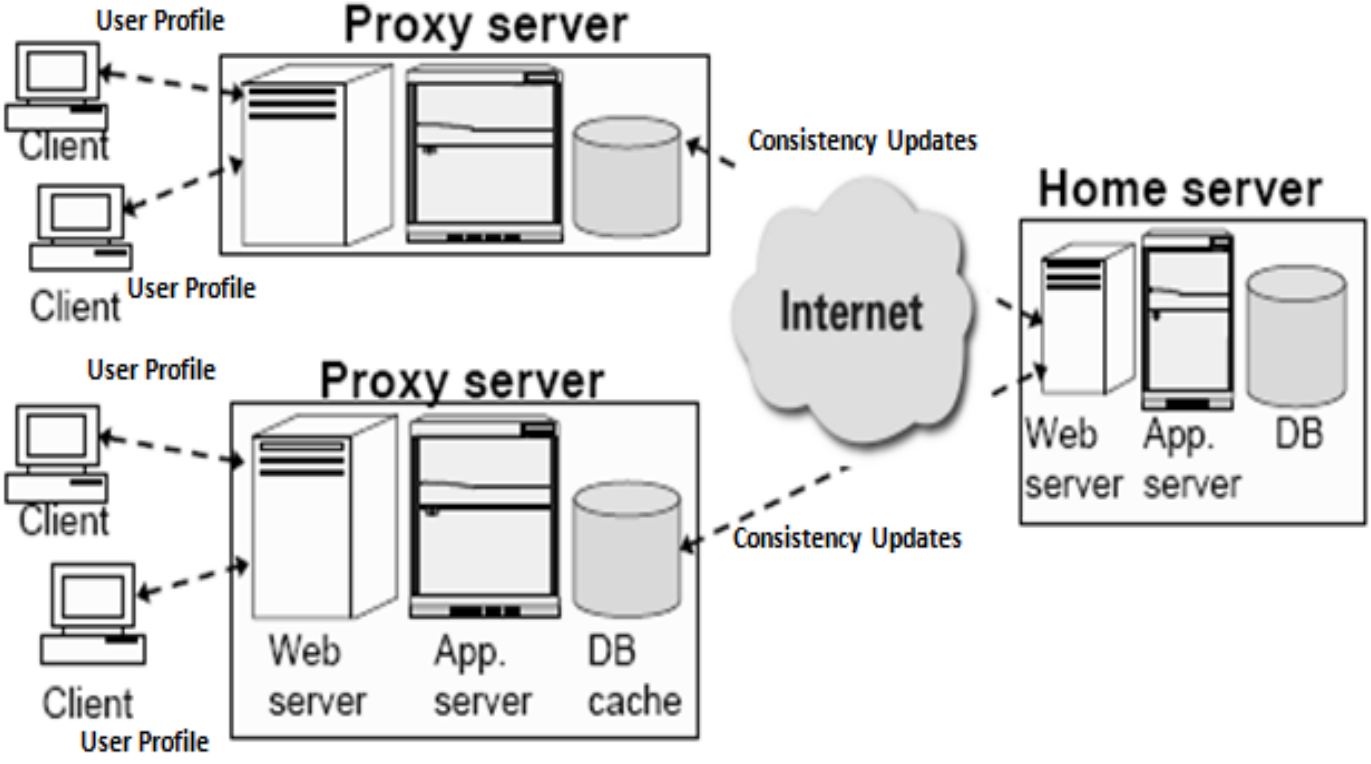

Figure 2: Envisioned Scalable Distributed Architecture for Web-based Applications

5. RESULT AND DISCUSSION

In the experimental setup, Apache Web server was ran on an Intel Pentium Dual Core Windows Vista machine with 2 gigabytes main memory. The proxies or clusters used for evaluating the performance of the proposed prototype have six computers connected to an Ethernet switch with 100Mbit/seconds network. All the nodes are identical. One

node was dedicated to being the front-end and another one was dedicated as the cache manager. In this experiment however, four backend nodes running one node manager process each was used. The proxy configuration employed for evaluation uses an event-driven architecture for all the components and a custom socket-based protocol for communicating between the components. The front-end uses the Replication LARD dispatcher and the node managers' caches use the GreedyDual-Size replacement policy which is an algorithm for uniform-size variable cost cache replacement. All tests have been run with the cluster with four backend nodes. Comparing the cluster's performance and scalability with a single-node server implemented using the PHP/MySQL platform allows for better understanding of the scalability issues on the cluster. The standalone Web server implements an event-driven model and sends response data using PHP's I/O facilities straight from the file-system cache of the operating system to the client. The Web server itself never handles directly or copies the bytes of the files that are served to clients.

A student table with fields such as student id, student name, matriculation number, gender, date of birth, telephone number, address, city, local government, religion, academic programme, department and session admitted, password as well as date registered was created in order to have a basis for comparisons. The developed system is a three-tier application that consists of the following layers:

a. ***Presentation layer*:** The presentation layer is responsible for all user interaction with the system. It accepts user input for forwarding to the application layer and displays application data to the user.

b. ***Application layer:*** The application layer encapsulates the system's business logic. It receives user requests from the presentation layer or from external clients via Web Services and may interact with the database layer in response to those requests.

c. ***Database layer*:** The database layer stores and retrieves queried data. Users interact with system via a browser using HTTP requests and responses. The presentation and application layers reside on one or more application servers. The database layer resides on a separate database server. Figure 3 shows a schematic view of the entire system.

The database was populated and the execution latencies of queries using the original *php* driver for different throughput values were obtained. In each case, the requested data is available via proxies and updated on the main server for future reference. For each run, three metrics were collected as follows:

a. Throughput; that is, the number of MBits per second that the server has served during the test.
b. Average response times; that is, how many seconds on average took the server to fulfill a request during the test.
c. Connections per second; that is, how many HTTP connections the server has been able to handle per second.

In order to ensure that each access is local, the server is assumed to be the master for all clusters. The execution latency which is the time taken by the server to generate the response for a request was computed and also, the overhead of the driver in processing the request was measured. The results of this experiment are visualized in Figure 4.

As seen in the Figure 4; even for sharp increase in the number of clients, the overhead introduced by the proposed system is between 1 and 2 seconds. This means that; when dynamic requests are used, the cluster shows good scalability thereby increasing connection rate significantly when backend nodes are added to the configuration. In other words, the connections per second suffer a decrease when comparing the standalone server with a one-backend cluster configuration, and the rate obtained with a four-backend cluster is over 300% higher than with the standalone server. It is also important to note that, with dynamic traffic (Figure 4), the cluster achieves considerably better (lower) response times than the standalone server when more than one backend node is used. The single-backend cluster configuration has slightly higher response times than the standalone server, but they decrease rapidly as backend nodes are added.

Results are very different for dynamic traffic; here the cluster with more than one backend node is able to outperform the single-node Web server in all metrics considered. Dynamic requests stress the Web serving system in a very different way than static requests. While for static requests pure I/O throughput of the system is the most important factor, dynamic traffic uses the Web server's CPU much more, and can benefit from the aggregated processing power available in a Web cluster. Performance of the standalone Web server is comparable or slightly higher than the one-backend cluster for dynamic traffic, but the cluster's performance increases significantly as backend nodes are added, providing much higher scalability for this kind of workload. Handling static and dynamic workloads require very different qualities from Web servers; static traffic handling is I/O-bound while dynamic traffic is more CPU-bound than I/O-bound. In I/O-bound applications optimizing the path of data and avoiding data copying are very important, and a Web cluster that uses a single front-end node and standard Ethernet network interconnections performs poorly in those aspects. However, when the majority of the traffic is dynamic and thus CPU-bound, the cluster's limitations from I/O processing performance point of view are a less important factor than the benefits obtained from the aggregated CPU power of the cluster.

We therefore conclude that, the overhead introduced by our driver is very low and negligible compared to the wide-area network latency incurred by traditional non-replicated systems.

International Journal of Information and Communication Technology Research

http://www.esjournals.org

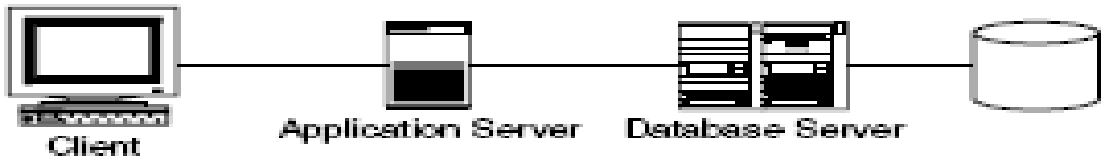

Figure 3: Schematic View of the system

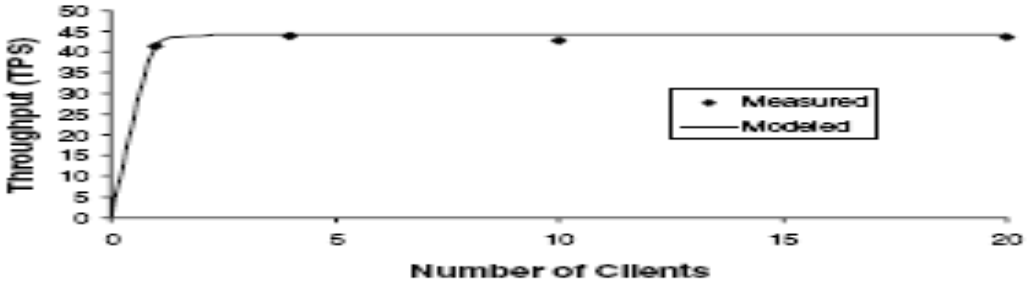

Figure 4: System Throughput

6. CONCLUSION

Most approaches toward scalable hosting of Web applications consider the application code and data structure as constants, and propose middleware layers to improve performance transparently to the application. This paper takes a different stand and demonstrates that major scalability improvements can be gained by allowing one to decentralize an application's data into independent services that are interoperable and allows regular updates using profile of users to respond to queries. While such restructuring introduces extra costs, it considerably simplifies the query access pattern that each service receives, and allows for a much more efficient use of classical scalability techniques.

REFERENCES

[1] Olston, C., Manjhi, A., Garrod, C., Ailamaki, A., Maggs, C., and Mowry, T., (2005). A scalability service for dynamic web applications. In Proc. Conf. on Innovative Data Systems Research.

[2] Cohen, (2001). DBProxy: A dynamic data cache for Web applications. In Proc. ICDE.

[3] Davis, A., Parikh, J., and Weihl, W., (2004). Edge computing: Extending enterprise applications to the edge of the internet. In Proc. www.

[4] DeCandia, G., Hastorum, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., (2007). Dynamo: Amazon's highly available key-value store. In Proc. SOSP.

[5] Cecchet, E., (2004). C-JDBC: a middleware framework for database clustering. Data Engineering, 27(2).

[6] Kemme, K., and Alonso, G., (2000). Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In Proc. VLDB.

[7] Plattner, C., and Alonso, G., (2004). Scalable replication for transactional web applications. In Proc. Middleware.

[8] Ronstrom, M., and Thalmann, L., (2004). MySQL cluster architecture overview. MySQL Technical White Paper, RUBBoS: Bulletin board system benchmark. http://jmob.objectweb.org/rubbos.html.

[9] Sivasubramanian, S., Pierre, G., Van Steen, M., and Alonso, G., (2007). Analysis of caching and replication strategies for web applications. IEEE Internet Computing, 11(1):60–66.

[10] Groothuyse, T., Sivasubramanian, S., and Pierre, G., (2007). GlobeTP: Template-based database replication for scalable web applications. In Proc. www.

[11] Gao, L., Dahlin, M ., Nayate, A., Zheng, J., and Iyengar, A., (2003). Application specific data replication for edge services. In Proc. www.

[12] Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R., (2006). Bigtable: A distributed storage system for structured data. In Proc. OSDI.

[13] Gribble, S., Brewer, E., Hellerstein, J., and Culler, D., (2000). Scalable, distributed data structures for internet service construction. In Proc. OSDI.

[14] Huang, Y., and Chen, J., (2001). Fragment in distributed database design. Information Science and Engineering, 17(3):491–506.

[15] Navathe, S., Karlapalem, K., and Ra, M.., (1995). A mixed fragmentation methodology for initial distributed database design. Computer and Software Engineering, 3(4).

[16] Kazerouni, L., and Karlapalem, K., (1997). Stepwise redesign of distributed relational databases. Technical Report 275 HKUST-CS97-12, Hong Kong Univ. of Science and Technology, Dept. of Computer Science.